ModProg 15-16, Vorl. 6

Richard Grzibovski

Nov. 25, 2015

Übersicht

1. Schleifesteuerung mit `break` und `continue`
2. `goto`-Befehl
3. Zeichen, Zeichenketten
4. Typumwandlungen
5. Formate für `printf`
6. Variablen (Teil 3)

Schleifesteuerung mit break

break-Befehl:

Mit Hilfe des `break`-Befehls wird die Schleife unabhängig von der Bedingung beendet und mit dem nächsten Befehl nach der Schleife fortgefahren.

Dies gilt für alle drei Schleifenformen:
`while`, `do ... while` und `for`.

Beispiel:

```
1  int finde_m_in_feld(int feld[], int lange, int m) {
2      int i;
3      for(i=0;i<lange;i++) {
4          if(feld[i]==m) break;     //      ---|
5      }                             //         |
6      return (i<lange)? i : -1;     // <------|
7  }
```

Schleifesteuerung mit continue

continue-Befehl:

Mit dem continue-Befehl wird zum Ende des *Schleifenkörpers* gesprungen.

- Bei `while`- und `do-while`-Schleifen wird nach `continue` die Bedingung abgefragt,
- bei `for`-Schleifen wird nach `continue` erst die Veränderung aufgerufen und dann die Bedingung abgefragt.

```
1  int zahle_geraden_in_feld(int feld[], int lange) {
2      int i, ger=0;
3      for(i=0;i<lange;i++) {
4          if(feld[i]%2) continue; //      ---|
5          ger++;                  //         |
6                                  //         |
7      }  //  <-------------------------------|
8      return ger;
9  }
```

goto-Befehl

Die Syntax lautet:

oder

```
1  Labelname:
2  ...
3  goto Labelname;
```

```
1  goto Labelname;
2  ...
3  Labelname:
```

Dabei ist `Labelname` ein beliebiger Name, der irgendwo im Programm (zumindest in der gleichen Funktion!) definiert sein muss.
Generell werden Programme mit `goto`-Befehlen sehr schnell unübersichtlich!

goto-Befehl

Ein **sehr gefährliches** Feature der goto-Anweisung ist die Möglichkeit, mitten in Blöcke springen zu können.

```
1  #include <stdio.h>
2  int main()
3  {
4      goto Label;
5      {
6          int Zahl = 10;
7  Label:
8          printf("Zahl = %i\n", Zahl);
9          /* uninitialisierte Zahl wird ausgegeben! */
10     }
11
12     return 0;
13 }
```

Datentyp Zeichen `char`

- Zeichen sind Buchstaben wie a, B, C, d, Ziffernzeichen wie 4, 5, 6 und Sonderzeichen wie ;,.! sowie andere Zeichen. Für sie gibt es den Datentyp `char`.
- Der Datentyp Zeichen beinhaltet immer nur ein Zeichen, das durch eine 1-Byte-Zahl (ganze Zahl) intern gespeichert wird.
- Daraus folgt, dass es 256 verschiedene Zeichen geben kann. Davon sind die ersten 128 international festgelegt, während die anderen 128 regional unterschiedlich sind. Der Zusammenhang zwischen den Zeichen und den intern gespeicherten Zahlen ist in der sogenannten ASCII-Tabelle festgelegt.
- Konstante Zeichen werden in Hochkommata eingeschlossen, also beispielsweise `'y'`, `'9'`, `'?'`.

Ein Teil aus ASCII-Tabelle

Zahl	Zeichen
47	/
48	0
49	1
50	2
51	3
52	4
53	5
54	6
55	7
56	8
57	9
58	:
59	;
60	<
61	=
62	>

Zahl	Zeichen
64	@
65	A
66	B
67	C
68	D
69	E
70	F
71	G
72	H
73	I
74	J
75	K
76	L
77	M
78	N
79	O

Zahl	Zeichen
97	a
98	b
99	c
100	d
101	e
102	f
103	g
104	h
105	i
106	j
107	k
108	l
109	m
110	n
111	o
112	p

Zeichen `char`

Es gibt besondere Zeichenkonstanten, die nicht direkt gedruckt bzw. auf dem Bildschirm angezeigt werden können. Um diese darzustellen, werden sie als zwei Zeichen geschrieben, benötigen aber wie alle Zeichen nur ein Byte.
Das erste der zwei Zeichen ist ein Backslash (\). Diese Zeichen werden auch *Escape-Sequenzen* genannt.

Zeichen	Bedeutung
`\a`	Signalton
`\f`	Seitenvorschub
`\n`	neue Zeile
`\r`	Zeilen- oder Wagenrücklauf
`\t`	Tabulator
`\\`	Backslash \
`\'`	Hochkomma '
`\"`	Anführungszeichen "
`\0`	Nullbyte (z.B. für String-Ende)
`\?`	Fragezeichen ? (C99)

Zeichen char

Da Zeichen intern als ganze Zahlen gespeichert werden, können alle Operatoren der ganzen Zahlen auch auf Zeichen angewendet werden, wobei nicht alle Operationen auch Sinn machen (beispielsweise die Addition zweier Zeichen).

Sinnvolle Operationen

Operator	Beispiel	Bedeutung
`=`	`a = 'X'`	Zuweisung
`<`	`a < 'z'`	kleiner als
`>`	`a > 'c'`	größer als
`<=`	`a <= 'M'`	kleiner als oder gleich
`>=`	`a >= 'k'`	größer als oder gleich
`==`	`a == 'J'`	gleich
`!=`	`a != 'n'`	ungleich

Beispiel

```
1  #include <stdio.h>
2  char mach_klein(char c) {
3      if(c>=65 && c<=90) return c+32;
4      if(c==196 || c==214 || c==220) return c+32;
5      // A uml      O uml      U uml
6      return c;
7  }
8  void alles_klein(char s[]) {
9      int i=0;
10     while(s[i]) s[i]=mach_klein(s[i++]);
11 }
12 int main(){
13    char str[]="Hallo Jorg!";
14    str[7]=246;
15    printf("%s\n",str);
16    alles_klein(str);
17    printf("%s\n",str);
18    return 0;
19 }
```

Konvertierung zwischen den Datentypen (Typumwandlung)

Die Syntax ist:

(Datentyp) Wert

Beispiel:

```
1  char c = 'A';
2  int i;
3  double s = 1.5;
4  i = (int) c; /* ergibt 65 */
5  i = 97;
6  c = (char) i; /* ergibt ein 'a' */
7  printf("%i\n", (int) s); /* druckt 1 */
```

Konvertierung zwischen den Datentypen (Typumwandlung)

Die Syntax ist:

```
(Datentyp) Wert
```

- Diese Konvertierung wird auch **explizite Konvertierung** bzw. **explizite Typumwandlung** genannt.
- Dabei muss aber berücksichtigt werden, dass sich nicht jeder Datentyp zu 100% konvertieren lässt; beispielsweise wenn eine Gleitpunktzahl in eine ganze Zahl konvertiert wird, gehen alle Nachkommastellen verloren.
- Durch die Konvertierung wird die Typkontrolle des Compilers umgangen und kann daher zu Fehlern führen.

Konvertierung zwischen den Datentypen (Typumwandlung)

Bei Operationen oder bei Funktionenaufruf vorkommen die **implizite Konvertierungen** bzw. **implizite Typumwandlungen**. Dabei wird der Datentyp, in den konvertiert werden soll, weggelassen.
Beispiel:

```
1  char c = 'A';
2  int i;
3  double s = 1.5;
4  i = c; /* ergibt 65 */
5  i = 97;
6  c = i; /* ergibt ein 'a' */
7  i = s; /* ergibt 1 */
8  printf("%i\n", s); /* <- Warnung von Compiler: */
9                     /*    falscher Typ          */
```

Hinweis: Bei der Konvertierung einer ganzen Zahl größer als 255 in ein Zeichen werden die überzähligen Bits nicht berücksichtigt.

Zeichenketten (Strings)

- Eine Zeichenkette (auch String genannt) ist ein Spezialfall eines Felds, nämlich ein eindimensionales Feld von Zeichen (`char`).
- In jedem Element des Felds wird genau ein Zeichen abgelegt. Alle Elemente zusammen ergeben die Zeichenkette, z.B. ein Satz.
- Damit eignen sich Zeichenketten zum Speichern von Texten, Meldungen, usw. Dabei kann mit dem Feld auf jeden Buchstaben einzeln zugegriffen werden.

Wichtig

Damit im Programm das Textende eindeutig erkannt werden kann, wird nach dem letzten Zeichen der Zeichenkette noch ein weiteres Element mit dem Zeichen `'\0'` (ASCII-Wert: 0) gespeichert. Das Feld muss also immer ein Element mehr haben als die Zeichenkette lang ist!!! Wird dieses nicht beachtet, wird das Programm fehlerhaft arbeiten oder sogar abstürzen!

Zeichenketten (Strings)

Beispiel:

```
char Text[] = {'D','i','e','s','','i','s','t',' ',
               'e','i','n',' ','T','e','x','t','!','\0'};
```

Damit wird ein Feld mit 19 Elementen (18 Textzeichen und 1 Abschlusszeichen) erzeugt.

Das obige Feld lässt sich auch einfacher initialisieren:

```
char Text[] = "Dies ist ein Text!";
```

Durch die Anführungsstriche setzt der Compiler automatisch das Abschlusszeichen an den Text heran.

Weiteres Beispiel:

```
1 while(Kette1[i])
2     Kette2[i] = Kette1[i++];
3 Kette2[i] = '\0'; /* Abschlusszeichen setzen */
```

Der Text von `Kette1` wird nach `Kette2` kopiert.

Datenausgabe auf den Bildschirm: die Funktion `printf`

- Die Standardfunktion für eine Bildschirmausgabe ist die `printf`-Funktion. Dabei steht printf für "print formatted".
- Um die Funktion verwenden zu können, muss die Headerdatei `stdio.h` mittels `#include` <stdio.h> eingebunden werden.

Die komplette Syntax der `printf`-Funktion lautet:

```
int printf(const char *format [,argument_1 ... ,argument_n]);
```

- Der Rückgabewert ist eine ganze Zahl (`int`), die angibt, wieviele Zeichen insgesamt ausgegeben wurden. Im Falle eines Fehlers ist das Ergebnis gleich dem Wert `EOF` (d.h. -1).
- `const char *format`: Der erste Parameter ist eine Zeichenkette. In dieser Zeichenkette stehen die Formatierungen für die auszugebenden Daten. Über die Formatierungsangaben wird auch die Anzahl der auszugebenden Daten festgelegt.
- `argument`: Nach der Formatierungsangabe folgen die Variablennamen, deren Inhalte auf dem Bildschirm ausgegeben werden sollen, jeweils getrennt mit einem Komma.

Die Formatierungsanweisungen

Die Formatierungszeichenkette beinhaltet zum einen "normale Zeichen", die direkt auf dem Bildschirm ausgegeben werden, und zum anderen die **Formatierungsanweisungen**. Eine Formatierungsanweisung hat folgenden Aufbau:

```
%[Flags][Breite][.Prazision][F|N|hh|h|l|ll|L]Typ
```

Jede Formatierungsanweisung beginnt mit einem Prozentzeichen.

`%[Flags][Breite][.Prazision][F|N|hh|h|l|ll|L]Typ`

Jede Formatierungsanweisung beginnt mit einem Prozentzeichen. Auf dieses Zeichen folgen:

- `[Flags]` Eine (optionale) Zeichenfolge, über die numerische Vorzeichen, Dezimalpunkte, führende und folgende Nullen, oktale und hexadezimale Präfixe sowie links- und rechtsbändige Ausgabe festgelegt werden.
- `[Breite]` Eine (optionale) Angabe über die minimal auszugebende Zeichenzahl. Notfalls wird mit Leerzeichen oder Nullen aufgefüllt.
- `[.Präzision]` Eine (optionale) Angabe, wieviele Zeichen maximal ausgegeben werden (Zeichenketten), die Minimalzahl von Ziffern (ganze Zahlen) bzw. die maximale Anzahl der Nachkommastellen (Gleitpunktzahlen).
- `[F|N|hh|h|l|ll|L]` Eine (optionale) Angabe der Größe des Parameters.
- `Typ` Die Angabe des Typs der auszugebenden Variable. Diese Angabe muss auf jeden Fall gemacht werden!

```c
#include <stdio.h>
int main()
{ int Zahl = 125;
 float Reel = 3033.1415f;
 char Zeichen = 'a', kette[]="abcdefg";
 printf("Textausgabe ohne Variablen\n");
 printf("Zahl hat den Wert %+05i\n", Zahl);
 printf("Zahl in hex. Darstellung: %#X\n", Zahl);
 printf("Reel hat den Wert %.10f\n", Reel);
 printf("Reel in Exponentendarst.: %015.3e\n", Reel);
 printf("Zeichen hat %-3c als Inhalt.\n", Zeichen);
 printf("Kette hat Inhalt %s \n", kette);
 return 0;
}
```

```
Textausgabe ohne Variablen
Zahl hat den Wert +0125
Zahl in hex. Darstellung: 0X7D
Reel hat den Wert 3033.1416015625
Reel in Exponentendarst.: 0000003.033e+03
Zeichen hat a   als Inhalt.
Kette hat Inhalt abcdefg
```

Variablen (Teil 3)

- Deklaration:
  `Speicherklasse` - `const` - `signed/unsigned` -
  `long/short Datentyp Variablenname;`
- Gültigkeitsbereich
- Sichtbarkeit/Überdeckung
- Speicherklassen:
  `auto` - für lokale Variable - sie existieren nur so lange, wie der entsprechende Block verarbeitet wird. Automatische globale Variablen existieren so lange wie das Programm und sind von allen Quelltext-Dateien des Programms aus erreichbar.
  `extern` - Kann für globale oder lokale Variablen verwendet werden. Durch dieses Schlüsselwort wird angegeben, dass die deklarierte Variable in einer anderen Quelltextdatei deklariert und definiert ist.
  `register` - Kann nur bei der Deklaration von lokalen Variablen oder bei Funktions-Parametern verwendet werden. Der Compiler bekommt damit den Hinweis, dass diese Variable häufig genutzt wird und dass die Variable "wenn möglich" so definiert werden sollte, dass die Zugriffszeit möglichst gering ist.

static

`static` - Kann für globale oder lokale Variablen verwendet werden. Statische Variablen sind wohl nur innerhalb des Blockes (lokal) bzw. innerhalb der Quelltextdatei (global) sichtbar, in denen sie deklariert wurden; sie sind dann aber bis zum Ende des Programms gültig! D.h. am Ende des Gültigkeitsbereiches von normalen Variablen werden statische Variablen nicht vernichtet und behalten sogar ihren Wert bei. Statische Variablen mit Initialisierung werden dadurch auch nur beim einmaligen Anlegen initialisiert; wird der Block mit der statischen Variablen-Deklaration erneut aufgerufen, existiert diese Variable bereits und die Initialsierung wird nicht durchgeführt. Wenn explicite Initialisierung ist nicht gegeben, ist die Variable mit Null initialisiert.

```c
#include <stdio.h>
int main()
{
    int i;
    for (i = 0; i < 5; i++)
    {
        static int Statisch = 0;
        int NichtStatisch = 0;
        Statisch = Statisch + 1;
        NichtStatisch = NichtStatisch + 1;
        printf("%i: Statisch = %d; ", i, Statisch);
        printf("NichtStatisch = %d\n", NichtStatisch);
    }
    return 0;
}
```

```
0: Statisch = 1; NichtStatisch = 1
1: Statisch = 2; NichtStatisch = 1
2: Statisch = 3; NichtStatisch = 1
3: Statisch = 4; NichtStatisch = 1
4: Statisch = 5; NichtStatisch = 1
```

```c
#include <stdio.h>
int func(int k);

int main()
{
    printf("5!=%i\n",func(5));
    printf("6!=%i\n",func(6));
    return 0;
}

int func(int k) {
    static int anz_aufrufe=0;
    anz_aufrufe++;
    printf("func laeft %5i Mal:",anz_aufrufe);
    printf("Argument ist %2i\n",k);
    if(k>1) return k*func(k-1);
    else    return 1;
}
```